Minimal Logic for Computable Functions

Helmut Schwichtenberg
Mathematisches Institut der Universität München
Theresienstr. 39, D-8000 München 2

We discuss a specification language with variables for higher order functionals and constants for computable functionals (cf. Plotkin [9]). In this language it is possible to represent functional objects (like programs or circuits transforming streams of data) by terms and carry out formal proofs that they meet a given specification.

The intended semantics is such that the domain of a computable functional consists of all continuous (partial) functionals in the sense of Scott [11]. We also allow non–monotone functionals (like equality or the property of being total, considered as boolean–valued functions) in our model; however, such functionals can only be applied to something and can never be arguments. In this sense they are external objects.

As our deduction system for this language we take the $\to\forall$–fragment of Gentzen's natural deduction (i.e. just introduction and elimination rules for $\to$ and $\forall$), plus induction schemata for natural numbers, boolean objects and finite functionals (cf. Scott [11]). $\neg, \exists, \vee$ are defined as usual. In spite of our minimal supply of logical rules we get classical logic, since we can derive the stability of atomic formulas, i.e. $\forall p\colon \mathrm{boole}(\neg\neg p \to p)$, by boolean induction (case analysis). Here we make use of the fact that we build atomic formulas from boolean terms.

We then extend this language by a strong existential quantifier written $\exists^*$ (as opposed to $\exists$ defined by $\neg\forall\neg$). A formula containing $\exists^*$ is generally not an entity the deduction system can deal with: some "realising terms" are required to turn it into a "judgement" (this terminology is due to Weyl [15] and has been taken up by Martin–Löf). E.g. r realises $\exists^* x \varphi(x)$ is a judgement, which can be translated into $\varphi(r)$ (cf. [12]).

Finally we use a recent implementation of this deduction system to give some examples. A main point here is that, since we only use the $\to\forall$–fragment of natural deduction, derivations are essentially terms in the typed λ–calculus extended by recursion constants. Hence it is possible to use the built–in evaluation mechanism of SCHEME (a LISP–dialect) to carry out the normalisation of proofs (a technical point here is that an inverse to the evaluation is needed to make this work; cf. [1]). This in turn makes it possible to use proofs as programs.

1. Specification

The aim of this paper is to discuss a framework for the formal verification of hardware and of functional programs. In this introductory section we want to demonstrate that a setting of minimal logic with constants denoting higher order primitive recursive functions and with the possibility to quantify over partial functions is well–suited to design and verify hardware components. The reason for this is that a piece of hardware transforms streams of (possibly undefined) data, and hence can adequately be modelled by a type two object. Moreover, a process to build a circuit from given components is to be modelled by a function taking type two arguments and is consequently of type three; however, we do not pursue this matter here.

Let us make these ideas more concrete and work out a simple example, the register (or D–flip–flop). Its specification is as follows. reg takes a control stream $\hat{c}$ and an input stream $\hat{\imath}$ and produces an output stream $\mathrm{reg}(\hat{c}, \hat{\imath})$, whose value at time $n+1$ should be the value of the input stream at the (unique) previous time point l with the property that the control stream at time l was true and from then on up to and including time n it was false. If no such time l exists, then the value is not specified.

Let us first write out this specification more formally. We will use the variables

$\hat{\imath}$	of type nat $\to$ word	for input streams
$\hat{c}$	of type nat $\to$ boole	for control stream
$\hat{w}$	of type word.	

possibly without the hat (i.e. c, i, w) to signalize that they range over total objects only, and with indices. Then the specification is

$$\forall \hat{c}, \hat{\imath}, n, l. l < n \to \hat{c}(l) = \mathrm{true} \to (\forall m. l < m < n \to \hat{c}(m) = \mathrm{false}) \to \mathrm{reg}(\hat{c}, \hat{\imath}, n) = \hat{\imath}(l). \tag{1}$$

We define reg by second order primitive recursion, as follows.

$$\begin{aligned} \mathrm{reg}(\hat{c}, \hat{\imath}, \bot^{\mathrm{nat}}) &= \bot^{\mathrm{word}} \\ \mathrm{reg}(\hat{c}, \hat{\imath}, 0) &= \bot^{\mathrm{word}} \\ \mathrm{reg}(\hat{c}, \hat{\imath}, n+1) &= \mathrm{if}(\hat{c}(n), \hat{\imath}(n), \mathrm{reg}(\hat{c}, \hat{\imath}, n)) \end{aligned}$$

From this definition we only need the last equation to prove (1). However, the full definition is needed to maintain the property that every closed term of a ground type normalizes into a canonical term of that ground type.

We now give an informal proof of (1), by induction on all total objects of type nat. Later in section 5.3 we will formalize this proof.

For $n = 0$ the claim is trivial since $l < 0$ is false. For $n+1$, assume the induction hypothesis, and furthermore

$$l < n+1,$$

$$\hat{c}(l) = \mathrm{true} \tag{2}$$

and

$$\forall m. l < m < n+1 \to \hat{c}(m) = \mathrm{false}. \tag{3}$$

We have to show that $\mathrm{reg}(\hat{c}, \hat{\imath}, n+1) = \hat{\imath}(l)$. Let us distinguish cases according to the possible values of $\hat{c}(n)$. Note that $\hat{c}$ is a partial function, thus $\bot^{\mathrm{boole}}$ is a possible value here.

Case $\hat{c}(n) = \bot^{\mathrm{boole}}$. We have to show $\bot^{\mathrm{boole}} = \hat{\imath}(l)$. From $l < n+1$ we can conclude that either $l < n$ or else $l = n$. The first case is impossible, since then form (3) we get $\hat{c}(n) = \mathrm{false}$ contradicting our case assumption, and the second case $l = n$ is impossible too since then (2) and our case assumption lead to a contradiction.

Case $\hat{c}(n) = \mathrm{true}$. We have to show $\hat{\imath}(n) = \hat{\imath}(l)$. From $l < n+1$ we can conclude that either $l < n$ or else $l = n$. The first case is impossible, since then form (3) we get $\hat{c}(n) = \mathrm{false}$ contradicting our case assumption, and in the second case $l = n$ the goal simply follows from an equality axiom.

Case $\hat{c}(n) = \mathrm{false}$. We have to show $\mathrm{reg}(\hat{c}, \hat{\imath}, n) = \hat{\imath}(l)$. By induction hypothesis it suffices to show $l < n$. Since $l < n+1$ we only have to exclude $l = n$. But $l = n$ is impossible since then (2) and our case assumption lead to a contradiction.

2. Denotational semantics

To give a meaning to terms involving constants like reg, we develop Gödel's notion of a primitive recursive function of higher type [6] in the context of partial continuous functionals as introduced by Kreisel in [7] and developed mainly by Scott and Yu. Ersov (see [11], [3]). To make this paper readable for people not familiar with the theory of partial continuous functionals we have included the introductory sections of [13]. We then give the well–known definition of totality for partial continuous functionals and a simple proof (essentially due to Longo and Moggi [8]) that the equivalence relation $\sim_\varrho$ on the set $\mathcal{T}_\varrho$ of total functionals of type ϱ defined by

$$z_1 \sim_{\varrho\to\sigma} z_2 \iff \forall x \in \mathcal{T}_\varrho (z_1 x \sim_\sigma z_2 x)$$

is in fact a congruence relation, i.e. compatible with application. Finally we discuss how our language can be extended by symbols for external, i.e. non–continous functionals, just as ordinary first order logic can be extended by adding function symbols. This is very useful for practice, since many functionals one wants to talk about (e.g. non–strict equality) are non–monotonic and hence non–continuous.

2.1 Finite functionals

The sets $|D_\varrho|$ of partial continuous functionals of type ϱ are the proper domains for computable functionals (Kreisel in [7] and Ersov in [3] give convincing arguments for this) and also for the partial primitive recursive functionals we want to study here. In section 2.2 we will give a definition of the sets $|D_\varrho|$, in a form convenient for our later arguments. The elements of $|D_\varrho|$, i.e. the partial continuous functionals of type ϱ, can be viewed as limits of certain finite functionals; such finite functionals are the subject of the present section. It seems best to treat them in the context of Scott's information systems of [11].

Definition 1. *An information system consists of a set D of (concrete) data objects, a set* Con *of finite subsets of D such that*

$$u \subseteq v \in \mathrm{Con} \implies u \in \mathrm{Con} \tag{1}$$

and for any $X \in D$

$$\{X\} \in \mathrm{Con}, \tag{2}$$

and a reflexive and transitive relation $\sqsupseteq$ on Con *such that for all $X_1, \dots, X_m \in D$ and $u \in \mathrm{Con}$*

$$u \sqsupseteq \{X_1, \dots, X_m\} \iff u \sqsupseteq \{X_1\} \wedge \dots \wedge u \sqsupseteq \{X_m\}. \tag{3}$$

Note that (3) implies that from $u \sqsupseteq v_1, \dots, u \sqsupseteq v_m$ we can conclude $v := v_1 \cup \dots \cup v_m \in \mathrm{Con}$ and $u \sqsupseteq v \sqsupseteq v_i$ for $i = 1, \dots, m$. The $u \in \mathrm{Con}$ are called *consistent finite sets of data objects*, or just (finite) *approximations*. $u \sqsupseteq v$ is read as "*u extends v*".

Our basic information system is D_{nat}, whose data objects are the natural numbers $0, 1, 2, \dots$, whose approximations are the singletons $\{0\}, \{1\}, \{2\}, \dots$ together with the empty set $\emptyset$, and whose extension relation $\sqsupseteq$ is just the set theoretic inclusion $\supseteq$. Similarly we construct the information system D_{boole} based on the boolean data objects false and true.

Given information systems D and E, we now construct a new information system $D \to E$, as in [11]. Its data objects are the pairs (u, v) with $u \in \mathrm{Con}_D$ and $v \in \mathrm{Con}_E$. A finite set $\{(u_i, v_i) \colon i \in I\}$ of data objects is consistent in $D \to E$ if

$$\forall I' \subseteq I (\bigcup_{i \in I'} u_i \in \mathrm{Con}_D \implies \bigcup_{i \in I'} v_i \in \mathrm{Con}_E). \tag{4}$$

In order to define the extension relation $\sqsupseteq$ for $D \to E$ we first define the result of an *application* of $W = \{(u_i, v_i) \colon i \in I\} \in Con_{D \to E}$ to $u \in \mathrm{Con}_D$:

$$\{(u_i, v_i) \colon i \in I\} u := \bigcup \{v_i \colon u \sqsupseteq u_i\}. \tag{5}$$

Then by (4) we know that $Wu \in \mathrm{Con}_E$. Obviously, application is monotone in the second argument, i.e.

$$u \sqsupseteq u' \implies Wu \sqsupseteq Wu'. \tag{6}$$

Now define $W \sqsupseteq W'$ by

$$W \sqsupseteq \{(u'_j, v'_j) \colon j \in J\} :\iff \forall j \in J. W u'_j \sqsupseteq v'_j. \tag{7}$$

Lemma 2. *If D and E are information systems, then so is $D \to E$.*

Proof. We first show the transitivity of $\sqsupseteq$. So let

$$W \sqsupseteq \{(u'_j, v'_j) \colon j \in J\} \sqsupseteq \{(u''_k, v''_k) \colon k \in K\}.$$

Then we have for all $k \in K$ by (6) and (7)

$$Wu''_k \sqsupseteq \bigcup\{Wu'_j \colon u''_k \sqsupseteq u'_j\} \sqsupseteq \bigcup\{v'_j \colon u''_k \sqsupseteq u'_j\} \sqsupseteq v''_k.$$

It remains to show (3) for $D \to E$. Since $\Rightarrow$ is obvious we only deal with $\Leftarrow$. So let $\{(u_i, v_i) \colon i \in I\} \sqsupseteq \{(u'_j, v'_j)\}$ for all $j \in J$. It suffices to show that $\{(u'_j, v'_j) \colon j \in J\}$ is consistent. So assume $J' \subseteq J$ and $\bigcup_{j \in J'} u'_j \in \mathrm{Con}_D$. By (4) we have to show that $\bigcup_{j \in J'} v'_j \in \mathrm{Con}_E$. But this follows from

$$\bigcup\{v_i \colon \bigcup_{j \in J'} u'_j \sqsupseteq u_i\} \sqsupseteq \bigcup\{v_i \colon u'_j \sqsupseteq u_i\} \sqsupseteq v'_j. \qquad \Box$$

Note that with the above definition of the extension relation $\sqsupseteq$ in $D \to E$ application is also monotone in the first argument, i.e.

$$W \sqsupseteq W' \implies Wu \sqsupseteq W'u. \tag{8}$$

To see this observe that

$$Wu \sqsupseteq \bigcup\{Wu'_j \colon u \sqsupseteq u'_j\} \sqsupseteq \bigcup\{v'_j \colon u \sqsupseteq u'_j\} = W'u.$$

We will exclusively deal with the information systems built up from D_{nat} and D_{boole} by the $\to$–operation. More formally, define the notion of a *type symbol* and its *level* inductively by the clauses

1. nat and boole are type symbols, and $\mathrm{lev}(\mathrm{nat}) = \mathrm{lev}(\mathrm{boole}) = 0$.
2. If ϱ and σ are type symbols, then so is $(\varrho \to \sigma)$, and

$$\mathrm{lev}(\varrho \to \sigma) = \max(\mathrm{lev}(\varrho) + 1, \mathrm{lev}(\sigma)).$$

As usual we write $\varrho_1, \ldots, \varrho_m \to \sigma$ for $(\varrho_1 \to (\varrho_2 \to \ldots (\varrho_m \to \sigma) \ldots))$. Note that any type symbol can be written uniquely in the form $\varrho_1, \ldots, \varrho_m \to \mathrm{nat}$ or $\varrho_1, \ldots, \varrho_m \to \mathrm{boole}$. For any type symbol ϱ define the information system D_ϱ as follows. D_{nat} and D_{boole} have already been defined, and $D_{\varrho \to \sigma} := D_\varrho \to D_\sigma$. The D_ϱ are called *standard information systems*.

Note that for standard information systems the exponential test (4) for consistency of a finite set of data objects can be replaced by a quadratic test. To see this call an information system *coherent* (see Plotkin [10, p. 210]) if for any finite set $\{X_i \colon i \in I\}$ of data objects

$$\forall i, j \in I. \{X_i, X_j\} \in \mathrm{Con} \implies \{X_i \colon i \in I\} \in \mathrm{Con}. \tag{9}$$

Obviously D_{nat} and D_{boole} are coherent. Now the coherence of all standard information systems D_ϱ follows from

Lemma 3. *If D and E are information systems and E is coherent, then so is $D \to E$.*

Proof. Let $\{(u_i, v_i) : i \in I\}$ be finite and assume

$$\forall i, j \in I. \{(u_i, v_i), (u_j, v_j)\} \in \mathrm{Con}_{D \to E}. \tag{10}$$

We have to show $\{(u_i, v_i) : i \in I\} \in \mathrm{Con}_{D \to E}$. So, by (4), assume $I' \subseteq I$ and $\bigcup_{i \in I'} u_i \in \mathrm{Con}_D$. We have to show $\bigcup_{i \in I'} v_i \in \mathrm{Con}_E$. Now since E is coherent by assumption, it suffices to show $v_i \cup v_j \in \mathrm{Con}_E$ for all $i, j \in I'$. So let $i, j \in I'$. By assumption we have $u_i \cup u_j \in \mathrm{Con}_D$ and hence by (10) and the definition of $\mathrm{Con}_{D \to E}$ also $v_i \cup v_j \in \mathrm{Con}_E$. $\square$

The elements of $\mathrm{Con}_\varrho := \mathrm{Con}_{D_\varrho}$, i.e. the consistent finite sets of data objects or approximations in D_ϱ will be called *finite functionals* of type ϱ. In section 2.2 they will be used to define the partial continuous functionals as limits of finite functionals. Finite functionals will also be special partial primitive recursive functionals.

2.2 Limits of finite functionals

We now give the definition (due to Scott [11]) of the partial continuous functionals of type ϱ, in a form suitable for our later arguments. They are taken as limits (or, more precisely, as ideals) of finite functionals.

Definition 1. *An ideal x in an information system D (written $x \in |D|$) is a set x of data objects which is consistent in the sense that any finite subset of x is in Con_D, and closed against $\sqsupseteq$, i. e. if $u \sqsupseteq \{X\}$ for some finite subset u of x, then $X \in x$.*

The crucial fact about ideals in $D \to E$ is that they can be identified with *continuous* functions from $|D|$ to $|E|$, defined as follows.

Definition 2. *Let $D_1, \ldots, D_m := \vec{D}$ and E be information systems. A function $f : |\vec{D}| \to |E|$ is called continuous if it is monotone, i.e. for all $\vec{x}, \vec{y} \in |\vec{D}|$*

$$\vec{x} \subseteq \vec{y} \implies f(\vec{x}) \subseteq f(\vec{y}) \tag{1}$$

and satisfies the approximation property, i.e. for all $\vec{x}, \vec{y} \in |\vec{D}|$ and $v \in \mathrm{Con}_E$

$$v \subseteq f(\vec{x}) \implies \exists \vec{u} \in \mathrm{Con}_{\vec{D}} (\vec{u} \subseteq \vec{x} \wedge v \subseteq f(\overline{\vec{u}})), \tag{2}$$

where $\overline{u}$ denotes the closure of u under $\sqsupseteq$, i.e. $\overline{u} := \{X : u \sqsupseteq \{X\}\}$.

It is well known that this notion of continuity is the same as the ordinary one with respect to the *Scott–topologies* of $|\vec{D}|$ and $|E|$, defined as follows. For any consistent (finite or infinite) set y of data objects in an information system D let

$$\check{y} := \{x \in |D| : x \supseteq y\}$$

Then $\{\check{u} : u \in \mathrm{Con}_D\}$ is the basis of a T_0–topology (the Scott–topology) on $|D|$, which has the properties

$$\exists u \in \mathrm{Con}_D . x = \overline{u} \iff \check{x} \text{ is open}$$

and

$$x \subseteq y \iff x \in \{y\}^- \iff \forall u \in \mathrm{Con}_D (x \in \check{u} \implies y \in \check{u}).$$

For the proofs we refer the reader to [11].

We now show how ideals in $D \to E$ can be considered as continuous functions from $|D|$ to $|E|$.

Lemma 3. *Let D and E be information systems. To any ideal $z \in |D \to E|$ we can associate a continuous function*

$$\mathrm{fct}_z\colon |D| \to |E|$$

by

$$\mathrm{fct}_z(x) := zx := \bigcup\{v : \exists u \subseteq x.(u,v) \in z\}. \tag{3}$$

Also, to any continuous function $f\colon |D| \to |E|$ we can associate an ideal

$$\mathrm{ideal}(f) \in |D \to E|$$

by

$$\mathrm{ideal}(f) := \{(u,v) : v \subseteq f(\overline{u})\}. \tag{4}$$

The assignments given by (3) and (4) are inverse to each other.

The proof is rather straightforward and not given here. □

The ideals $x \in |D_\varrho|$ are called *partial continuous functionals* of type ϱ. Application of $z \in |D_{\varrho\to\sigma}|$ to $x \in |D_\varrho|$ is given by (3). Note that this application operation zx is continuous in both arguments, in the sense of Definition 2.

An important consequence of the identification of ideals $z \in |D \to E|$ with continuous functions from $|D|$ to $|E|$ given in Lemma 3 is the following *extensionality* property

Lemma 4. *Let D and E be information system and $z, z' \in |D \to E|$. Then from $z\overline{u} \subseteq z'\overline{u}$ for all $u \in \mathrm{Con}_D$ we can conclude $z \subseteq z'$.*

Proof. $z = \mathrm{ideal}(\mathrm{fct}_z) = \{(u,v) : v \subseteq z\overline{u}\}$. □

Hence the sets $|D_\varrho|$ of partial continuous functionals together with the application operators given by (3) form a *pre–structure* in the sense of Friedman [4, p. 23]. (Statman calls this a *frame* in [14, p. 331])

Any $z \in |D \to (E \to F)|$ can be viewed as a binary function $\mathrm{fct}^2_z : |D|, |E| \to |F|$ defined by

$$\mathrm{fct}^2_z(x,y) := \mathrm{fct}_{\mathrm{fct}_z(x)}(y).$$

We want to characterize the functions which can be obtained in this way. It turns out that these are exactly the binary continuous functions in the sense of Definition 2. This is a consequence of Lemma 3 together with the following

Lemma 5. *Let $D_1, \dots, D_m, E$ and F be information systems. To any continuous function $f\colon |D_1|, \dots, |D_m|, |E| \to |F|$ we can associate a continuous*

$$f_-\colon |D_1|, \dots, |D_m| \to |E \to F|$$

by

$$f_-(x_1, \dots, x_m) = \mathrm{ideal}(f(x_1, \dots, x_m, \cdot)), \tag{5}$$

where $f(x_1, \dots, x_m, \cdot)\colon |E| \to |F|$ is defined by $f(x_1, \dots, x_m, \cdot)(y) = f(x_1, \dots, x_m, y)$. Also to any continuous $g : |D_1|, \dots, |D_m| \to |E \to F|$ we can associate a continuous

$$g_+\colon |D_1|, \dots, |D_m|, |E| \to |F|$$

by

$$g_+(x_1, \dots, x_m, y) = \mathrm{fct}_{g(x_1,\dots,x_m)}(y). \tag{6}$$

The assignments given by (5) and (6) are inverse to each other.

Proof. Monotonicity and the approximation property can be verified easily, for f_- as well as for g_+. Furthermore, we have

$$(f_-)_+(\vec{x}, y) = \mathrm{fct}_{f_-(\vec{x})}(y) = \mathrm{fct}_{\mathrm{ideal}(f(\vec{x},\cdot))}(y) = f(\vec{x}, y)$$

and

$$(g_+)_-(\vec{x}) = \mathrm{ideal}(g_+(\vec{x}, \cdot)) = \mathrm{ideal}(\mathrm{fct}_{g(\vec{x})}) = g(\vec{x}),$$

where, in both cases, the last equation follows from Lemma 3. □

We now show that the sets $|D_\varrho|$ of partial continuous functionals together with the application operators (3) form a *model* of the typed λ–calculus (or a *structure*, in the terminology of Friedman [4, p. 23]).

The *terms*, their *types* and their sets of *free variables* are given by

1. Any variable x_i^ϱ $(i = 0, 1, 2, \dots)$ is a term of type ϱ, $\mathrm{FV}(x_i^\varrho) = x_i^\varrho$.
2. If r is a term of type σ, then $\lambda x_i^\varrho.r$ is a term of type $\varrho \to \sigma$, $\mathrm{FV}(\lambda x_i^\varrho.r) = \mathrm{FV}(r) \setminus \{x_i^\varrho\}$.
3. If t is a term of type $\varrho \to \sigma$ and s is a term of type ϱ, then (ts) is a term of type σ, $\mathrm{FV}((ts)) = \mathrm{FV}(t) \cup \mathrm{FV}(s)$.

We write $ts_1s_2 \dots s_m$ for $(\dots((ts_1)s_2)\dots s_m)$, and $\lambda x_1x_2 \dots x_m.r$ for $\lambda x_1.\lambda x_2 \dots \lambda x_m.r$. A term is called *closed* if $\mathrm{FV}(r) = \emptyset$.

For any term r of type σ and any list $\vec{x}$ of variables of types $\vec{\varrho}$ containing all the variables free in r we define a continuous function

$$|\vec{x} \mapsto r| \colon |D_{\varrho_1}|, \dots, |D_{\varrho_m}| \to |D_\sigma|$$

by induction on r, as follows.

1. $|\vec{x} \mapsto x_i|$ is the i–th projection function, which is clearly continuous.
2. $|\vec{x} \mapsto \lambda y.r| := |\vec{x}, y \mapsto r|_-$ (cf. Lemma 5).
3. $|\vec{x} \mapsto ts|$ is the result of substituting the functions $|\vec{x} \mapsto t|$ and $|\vec{x} \mapsto s|$ in the continuous binary application function. Clearly the resulting function is continuous.

Now we can define the *value* of a term r with free variables among $\vec{x}$ under an assignment of partial continuous functionals $\vec{\mathbf{x}}$ to the variables $\vec{x}$ to be just $|\vec{x} \mapsto r|\vec{\mathbf{x}}$. In particular, for any closed term r of type ϱ we have defined its value $|r| \in |D_\varrho|$.

2.3 Primitive recursion

There are far more continuous functions $f\colon |D_{\vec{\varrho}}| \to |D_\sigma|$ than just those given by terms r of the typed λ–calculus. Of special importance (and much studied in the literature, e.g. in [11], [3], [9]) are the *computable* ones, which by definition are those given by recursively enumerable ideals of finite functionals (in $|D_{\vec{\varrho}\to\sigma}|$). Here we want to deal with the more restricted (and hence better to analyze) notion of a primitive recursive functional, due to Gödel. Now what are primitive recursive functions $f\colon |D_{\vec{\varrho}}| \to |D_\sigma|$? It seems best to define them by means of an extension of the notion of a term in the typed λ–calculus.

For any type symbol ϱ, let countably many variables x_i^ϱ be given. Also, for any finite functional $u \in \mathrm{Con}_\varrho$, we introduce a constant $[u]^\varrho$. The constant $[m]^{\mathrm{nat}}$ (denoting the numeral m) is abbreviated by m^{nat} or just m, and the constant $[\emptyset]^\varrho$ (denoting the totally undefined finite functional in Con_ϱ) is abbreviated by $\perp^\varrho$. We further introduce a constant N for the successor function. Finally, for any type symbol ϱ, we introduce a recursion constant R_ϱ. The *primitive recursive terms*, their *types* and their sets of *free variables* are given by

1a. x_i^ϱ is a primitive recursive term of type ϱ, $\mathrm{FV}(x_i^\varrho) = x_i^\varrho$.

1b. $[u]^\varrho$ is a primitive recursive term of type ϱ, $\mathrm{FV}([u]^\varrho) = \emptyset$.

1c. N is a primitive recursive term of type $\mathrm{nat} \to \mathrm{nat}$, $\mathrm{FV}(N) = \emptyset$.

1d. R_ϱ is a primitive recursive term of type $\mathrm{nat}, \varrho, (\mathrm{nat}, \varrho \to \varrho) \to \varrho$, $\mathrm{FV}(R_\varrho) = \emptyset$.

2. If r is a primitive recursive term of type σ, then $\lambda x_i^\varrho.r$ is a primitive recursive term of type $\varrho \to \sigma$, $\mathrm{FV}(\lambda x_i^\varrho.r) = \mathrm{FV}(r) \setminus \{x_i^\varrho\}$.
3. If t is a primitive recursive term of type $\varrho \to \sigma$ and s is a primitive recursive term of type ϱ, then ts is a primitive recursive term of type σ, $\mathrm{FV}(ts) = \mathrm{FV}(t) \cup \mathrm{FV}(s)$.

In order to define the *value* of a primitive recursive term we first have to define a value $|R_\varrho| \in |D_{\mathrm{nat},\varrho,(\mathrm{nat},\varrho\to\varrho)\to\varrho}|$ for each recursion constant R_ϱ. This can be done as follows. For any nonnegative integer m, define a function

$$h_m\colon |D_\varrho|, |D_{\mathrm{nat},\varrho\to\varrho}| \to |D_\varrho|$$

by

$$h_0(y,z) = y$$
$$h_{m+1}(y,z) = z\{m\}(h_m(y,z)).$$

Clearly each h_m is continuous, by induction on m. Now define a function

$$f\colon |D_{\mathrm{nat}}| \to |D_{\varrho,(\mathrm{nat},\varrho\to\mathrm{nat})\to\varrho}|$$

by

$$f(\emptyset) = \overline{\emptyset}$$
$$f(\{m\}) = (h_m)_{--},$$

using Lemma 5 of the previous subsection. Obviously f is continuous. Finally let $|R_\varrho| := f_-$. We then have $|R_\varrho|_{+++}(x,y,z) = f_{++}(x,y,z)$, and from the definition of f

we obtain

$$\begin{aligned}
|R_\varrho|_{+++}(\emptyset, y, z) &= \overline{\emptyset}, \\
|R_\varrho|_{+++}(\{0\}, y, z) &= h_0(y, z) = y, \\
|R_\varrho|_{+++}(\{m+1\}, y, z) &= h_{m+1}(y, z) = z\{m\}(|R_\varrho|_{+++}(\{m\}, y, z)).
\end{aligned}$$

Exactly as for terms of the typed λ–calculus (in section 2.2) we can now define, for any primitive recursive term r of type σ and any list $\vec{x}$ of variables of types ϱ containing all the variables free in r, a continuous function $|\vec{x} \mapsto r|\colon |D_{\varrho_1}|, \dots, |D_{\varrho_m}| \to |D_\sigma|$, by induction on r. Just add, in clause 1, that $|\vec{x} \mapsto [u]^\varrho|$ is the constant function with value $\overline{u} \in |D_\varrho|$, $|\vec{x} \mapsto N|$ is the constant function with value the successor function $\in |D_{\mathrm{nat}\to\mathrm{nat}}|$ (which is clearly continuous), and $|\vec{x} \mapsto R_\varrho|$ is the constant function with value $|R_\varrho| \in |D_{\mathrm{nat},\varrho,(\mathrm{nat},\varrho\to\varrho)\to\varrho}|$ defined above.

Now we define the *value* of a primitive recursive term r with the free variables among $\vec{x}$ under an assignment of partial continuous functionals $\mathbf{x}$ to the variables $\vec{x}$ to be just $|\vec{x} \mapsto r|\mathbf{x}$. In particular, for any closed primitive recursive term r of type ϱ we have defined its value $|r| \in |D_\varrho|$. Let

$$|D_\varrho|^{pr} := \{|r|\colon r \text{ closed primitive recursive term of type } \varrho\} \subseteq |D_\varrho|.$$

The elements of $|D_\varrho|^{pr}$ are called *partial primitive recursive functionals*. Note that any element of $|D_{\varrho\to\sigma}|^{pr}$ can be viewed — via Lemma 5 of the previous subsection — as a continuous function $|D_\varrho| \to |D_\sigma|$.

It seems worthwile to also note that any partial primitive recursive functional when viewed as a function $f\colon |D_\varrho| \to |D_\sigma|$ is defined on all of $|D_\varrho|$, i.e. on all partial continuous functionals of type ϱ, not just on the subset $|D_\varrho|^{pr}$. of type ϱ. This seems to be desirable, since e.g. a primitive recursive operation on the reals like the exponential function e^x should be defined on arbitrary Cauchy sequences of rationals, not just on the primitive recursive ones.

The sets $|D_\varrho|^{pr} \subseteq |D_\varrho|$ are closed against application, since the primitive recursive terms are. Now consider any system $\{\mathfrak{M}_\varrho\}$ of sets satisfying $|D_\varrho|^{pr} \subseteq \mathfrak{M}_\varrho \subseteq |D_\varrho|$ and closed against application. By Lemma 4 of the previous subsection we know that the extensionality condition holds for $\{\mathfrak{M}_\varrho\}$, i.e. if $x, y \in \mathfrak{M}_{\varrho\to\sigma}$ and $\forall z \in \mathfrak{M}_\varrho(xz = yz)$ then $x = y$. Hence the sets $\mathfrak{M}_\varrho$ form a pre–structure in the sense of Friedman. They also form a model of the typed λ–calculus, since the $|D_\varrho|^{pr}$ do. We view such structures $\{\mathfrak{M}_\varrho\}$ as the intended models of theories involving primitive recursive terms.

Any primitive recursive term r can be transformed into a normal form r^* with the same value and such that this normal form is a constant provided the original term is closed and of a ground type. This is not completely obvious, since the presence of constants $[\{(u_i, v_i)\colon i \in I\}]$ for finite functionals creates some difficulties. We must be able to convert e.g. $[\{(u_i, v_i)\colon i \in I\}]r$, and the result should have as its value the supremum of all v_i with i such that the value of r extends u_i. In order to deal with this difficulty we must extend our notion of a primitive recursive term by some more term–forming operations. For the details we refer the reader to [13].

2.4 Total functionals

By induction on the type ϱ we define when a function of type ϱ is to be called *total*: $x \in |D_{\mathrm{nat}}|$ is total if it is of the form $\{m\}$, $x \in |D_{\mathrm{boole}}|$ is total if it is of the form $\{\mathrm{true}\}$ or $\{\mathrm{false}\}$, $z \in |D_{\varrho\to\sigma}|$ is total if for any total $x \in |D_\varrho|$ the value $zx \in |D_\sigma|$ is also total.

Frequently the total functions are the only ones we are really interested in (but still we need the partial ones, since any total function of type $\varrho \to \sigma$ has $|D_\varrho|$ as its domain). E.g., a real is a total Cauchy sequence of rationals.

The sets $\mathcal{T}_\varrho$ of total functions together with the application operations can be made into a pre–structure, by dividing them through the following equivalence relation. Let

$$\begin{aligned}
\{m\} \sim_{\mathrm{nat}} \{n\} &\iff m = n \\
\{p\} \sim_{\mathrm{boole}} \{q\} &\iff p = q \\
z_1 \sim_{\varrho\to\sigma} z_2 &\iff \forall x \in \mathcal{T}_\varrho . z_1 x \sim_\sigma z_2 x
\end{aligned}$$

We must show that the $\sim_\varrho$ are in fact a congruence relation, i.e. compatible with application. We prove that by a simple argument essentially due to Longo and Moggi [8]. First we need an auxiliary lemma, which says that with z_1 and z_2 also $z_1 \cap z_2$ (the intersection of the ideals) is total.

Lemma. $\mathrm{fct}_{z_1\cap z_2}(x) = \mathrm{fct}_{z_1}(x) \cap \mathrm{fct}_{z_2}(x)$.

Proof. This follows easily from the definition of $\mathrm{fct}_z(x)$. We have

$$\begin{aligned}
\mathrm{fct}_{z_1\cap z_2}(x) &= \bigcup\{v : \exists u \subseteq x : (u, v) \in z_1 \cap z_2\} \\
\mathrm{fct}_{z_i}(x) &= \bigcup\{v : \exists u \subseteq x : (u, v) \in z_i\}
\end{aligned}$$

Then $\subseteq$ is immediate, since any v from the upper union occurs (with the same u) in both lower unions. For $\supseteq$, let $X \in v_i$, $u_i \subseteq x$ and $(u_i, v_i) \in z_i$ for $i = 1, 2$. Let $u := u_1 \cup u_2 \subseteq x$. Then also $(u, \{X\}) \in z_i$. □

Lemma. *For total x, y, z we have*

$$x \sim_\varrho y \implies zx \sim_\sigma zy$$

Proof. We first show that from $x \sim_\varrho y$ we can conclude that $x \cap y$ is total. This is true since by the last lemma, for total $\vec{z}$ of type $\vec{\rho}$ where $\rho = \vec{\rho} \to \tau$ with $\mathrm{lev}(\tau) = 0$,

$$(x \cap y)\vec{z} = x\vec{z} \cap y\vec{z} = x\vec{z} = y\vec{z}.$$

Then we obtain, for total $\vec{z}$,

$$zx\vec{z} = z(x \cap y)\vec{z} = zy\vec{z}$$

and hence $zx \sim_\sigma zy$. □

A continuous function $f : |D_{\varrho_1}|, \dots, |D_{\varrho_m}| \to |D_\sigma|$ is called *total* if it maps total arguments into a total value. It follows easily from the definitions that with f also f_- and f_+ from section 2.2 are total.

Examples of total functions are $|R^{\mathrm{nat}}_\varrho|$ and $|R^{\mathrm{boole}}_\varrho|$. Hence for any primitive recursive term r built from these, from N and only total constants $[u]$ we have that $|\vec{x} \mapsto r|\vec{\mathbf{x}}$ is total provided the $\vec{\mathbf{x}}$ are.

2.5 External functionals

We now extend our language by symbols for *external*, i.e. non–continuous functions. Examples are definedness

$$\delta_{\mathrm{nat}}\colon D_{\mathrm{nat}} \to D_{\mathrm{boole}}$$

$$\delta_{\mathrm{nat}}(x) = \begin{cases} \text{true}, & \text{if } x \text{ is defined;} \\ \text{false}, & \text{otherwise.} \end{cases}$$

and non-strict equality

$$=_{\mathrm{nat}}\colon D_{\mathrm{nat}} \to D_{\mathrm{nat}} \to D_{\mathrm{boole}},$$

and similarly δ_{boole} and $=_{\mathrm{boole}}$. This is to be done in such a way that the semantics is not changed. We clearly have to restrict the syntax then, since unrestricted use of λ–abstraction, e.g. in $\lambda x.x = 0$, would lead to terms without a value (since only contimuous functions can be values).

Let $\mathfrak{F}$ be a set of function symbols of "functionality"

$$f\colon (\varrho_1, \ldots, \varrho_m) \to \sigma.$$

The $\mathfrak{F}$–*terms*, their *types*, their sets of *free variables* and their sets of *nonabstractable variables* are given by

1a. x_i^ϱ is an $\mathfrak{F}$–term of type ϱ, $\mathrm{FV}(x_i^\varrho) = x_i^\varrho$, $\mathrm{nonabs}(x_i^\varrho) = \emptyset$.

1b. $[u]^\varrho$ is an $\mathfrak{F}$–term of type ϱ, $\mathrm{FV}([u]^\varrho) = \mathrm{nonabs}([u]^\varrho) = \emptyset$.

1c. N is an $\mathfrak{F}$–term of type nat $\to$ nat, $\mathrm{FV}(N) = \mathrm{nonabs}(N) = \emptyset$.

1d. R_ϱ is an $\mathfrak{F}$–term of type nat, ϱ, (nat, $\varrho \to \varrho) \to \varrho$, $\mathrm{FV}(R_\varrho) = \mathrm{nonabs}(R_\varrho) = \emptyset$.

2. If r is an $\mathfrak{F}$–term of type σ and $x \notin \mathrm{nonabs}(r)$, then $\lambda x_i^\varrho.r$ is an $\mathfrak{F}$–term of type $\varrho \to \sigma$, $\mathrm{FV}(\lambda x_i^\varrho.r) = \mathrm{FV}(r) \setminus \{x_i^\varrho\}$, $\mathrm{nonabs}(\lambda x_i^\varrho.r) = \mathrm{nonabs}(r)$.
3. If t is an $\mathfrak{F}$–term of type $\varrho \to \sigma$ and s is an $\mathfrak{F}$–term of type ϱ, then ts is an $\mathfrak{F}$–term of type σ, $\mathrm{FV}(ts) = \mathrm{FV}(t) \cup \mathrm{FV}(s)$. $\mathrm{nonabs}((ts)) = \mathrm{nonabs}(t) \cup \mathrm{nonabs}(s)$,
4. (Function application) If $f \in \mathfrak{F}$ is a function symbol of functionality $(\varrho_1, \ldots, \varrho_m) \to \sigma$ and $r_1, \ldots, r_m$ are $\mathfrak{F}$–terms of types $\varrho_1, \ldots, \varrho_m$, then $fr_1 \ldots r_m$ is an $\mathfrak{F}$–term of type σ, $\mathrm{FV}(fr_1 \ldots r_m) = \mathrm{nonabs}(fr_1 \ldots r_m) = \mathrm{FV}(r_1) \cup \cdots \cup \mathrm{FV}(r_m)$.

Note that f is not an $\mathfrak{F}$–term. A variable x is called *abstractable* in r if $x \notin \mathrm{nonabs}(r)$.

In order to give a semantics for $\mathfrak{F}$–terms, we must assume that we have a (possibly non–continuous)

$$\mathbf{f}\colon |D_{\varrho_1}|, \ldots, |D_{\varrho_m}| \to |D_\sigma|$$

for any $f\colon (\varrho_1, \ldots, \varrho_m) \to \sigma \in \mathfrak{F}$. We now define, for any $\mathfrak{F}$–term r of type σ, any list $\vec{x}$ of variables of types $\vec{\varrho}$ containing all abstractable but no nonabstractable free variables of r (i.e. $\{\vec{x}\} \supseteq \mathrm{FV}(r) \setminus \mathrm{nonabs}(r)$, $\{\vec{x}\} \cap \mathrm{nonabs}(r) = \emptyset$) and any assignment η of continuous functions to the nonabstractable variables in r, a continuous function

$$|\vec{x} \mapsto r|\eta\colon |D_{\varrho_1}|, \ldots, |D_{\varrho_m}| \to |D_\sigma|$$

by induction on r. This definition is very similar to the corresponding definition for terms of the typed λ–calculus given in section 2.2, so we only treat the clauses for λ–abstraction and function application.

2. Let r be an $\mathfrak{F}$–term and $x \notin \mathrm{nonabs}(r)$. Then $|\vec{x} \mapsto \lambda x.r|\eta := |\vec{x}, x \mapsto r|_{-}\eta$.
4. Let $fr_1 \dots r_m$ with $f \in \mathfrak{F}$ be an $\mathfrak{F}$–term. Since $fr_1 \dots r_m$ has no abstractable variables, $|\vec{x} \mapsto fr_1 \dots r_m|\eta$ is defined to be a constant function whose value is given as follows. Let $\vec{x}_i$ be a list of all abstractable free variables in r_i (i.e. $\mathrm{FV}(r_i) \setminus \mathrm{nonabs}(r_i)$), $\vec{y}_i$ be a list of all nonabstractable free variables in r_i (i.e. $\mathrm{nonabs}(r_i)$) and η_i be the restriction of η to $\vec{y}_i$. Then the desired value is

$$\mathbf{f}((|\vec{x}_1 \mapsto r_1|\eta_1)\eta(\vec{x}_1)) \dots ((|\vec{x}_m \mapsto r_m|\eta_m)\eta(\vec{x}_m))$$

Note that $\mathbf{f}$ is generally "external", i.e. there need not be an $a \in |D_{\varrho_1 \to \dots \to \varrho_m \to \sigma}|$ such that

$$ab_1 \dots b_m = \mathbf{f}(b_1, \dots, b_m)$$

for all $\vec{b}$.

Note also that the set of $\mathfrak{F}$–terms is closed against substitution.

3. Logic

We now describe a formal system of higher order arithmetic, with the domains $|D_\rho|$ of higher order continuous functions as its intended model. The ground types are (at least) nat and boole, so we require induction axioms for both types.

We think of an atomic formula as being given by a boolean term. Hence, by boolean induction, we can prove stability $\neg\neg A \to A$ for any atomic formula A, and from this we can conclude the stability $\neg\neg\varphi \to \varphi$ of an arbitrary formula φ built from atoms by $\to$ and $\forall$, by a simple (meta–) induction on φ, requiring only introduction and elimination rules for $\to$ and $\forall$. Since falsity $\bot$ is present (the atom given by the boolean constant false), we can define negation $\neg\varphi$, disjunction $\varphi \vee \psi$ and the existential quantifier $\exists x\varphi$ as usual. In this way we get the strength of classical logic in spite of the fact that we only have the rules $\to^+, \to^-, \forall^+, \forall^-$ of minimal logic as our logical basis. This in turn makes it possible to represent proofs as λ–terms (with $\to^+, \forall^+$ corresponding to abstraction and $\to^-, \forall^-$ corresponding to application) with constants for induction axioms, hence the normalization theorem can be proved by a straightforward extension of the argument for primitive recursive terms (cf. section 2.3).

Since we want to deal with computable higher order functions it is appropriate to let our quantifiers range over over the partial continuous functions (cf section 2). On the other hand, in practice one often is interested in total functions only. Thus we must be able to express in our language the totality of a function. Using the idea of "external" functions discussed in section 2.5 we can do that easily: we first introduce a constant for an external function $\delta_{\mathrm{nat}} \colon D_{\mathrm{nat}} \to D_{\mathrm{boole}}$ defined by

$$\delta_{\mathrm{nat}}(x) = \begin{cases} \text{true}, & \text{if } x \text{ is defined;} \\ \text{false}, & \text{otherwise,} \end{cases}$$

and then use this constant to define totality of an arbitrary function of type ρ as in section 2.4.

However, from the practical point of view of readability of formulas it is a nuisance to be forced to restrict quantifiers any time one wants to talk about total functions.

Hence we use two sorts of variables for any type ρ, one — written $\hat{x}^\rho$ — for arbitrary functions and one — written x^ρ — for total functions of type ρ.

Let us now formally introduce the language. For any type ρ, we have infinitely many *variables* $\hat{x}_0^\rho, \hat{x}_1^\rho, \hat{x}_2^\rho, \ldots$ intended to range over arbitrary continuous functions and $x_0^\rho, x_1^\rho, x_2^\rho, \ldots$ intended to range over total continuous functions. We assume that a set $\mathfrak{C}$ of *program constants* is given, each of an arbitrary type ρ. They are intended to denote special primitive recursive functionals. The constants $[u]^\rho$, N and R_ρ would suffice here; however, it is useful in practice to also allow other constants. Furthermore, we assume that a set $\mathfrak{F}$ of *function symbols* is given, each of a "functionality" $(\rho_1, \ldots, \rho_m) \to \sigma$. They are intended to denote external, i.e. non–continuous functionals. We always require that the function symbols $\delta_{\mathrm{nat}}, \delta_{\mathrm{boole}}$ for definedness and $=_{\mathrm{nat}}, =_{\mathrm{boole}}$ for non–strict equality are available, as defined in section 2.5. We then define $\mathfrak{F}$–terms and their values, just as in section 2.5.

In order to define the normal form of $\mathfrak{F}$–terms, we must state conversion rules for all program constants and function symbols of our language. E.g. for δ_{nat} we have that $\delta(\bot^{\mathrm{nat}})$ converts into false and for any "total" term r (see below) $\delta(r)$ converts into true. Similarly, for $=_{\mathrm{boole}}$ we have that

$$
\begin{array}{rlll}
r = r & \text{converts into} & \text{true}, & \\
\bot^{\mathrm{boole}} = s & \text{converts into} & \text{false} & \text{if } s \text{ is total}, \\
r = \bot^{\mathrm{boole}} & \text{converts into} & \text{false} & \text{if } r \text{ is total}, \\
\text{true} = \text{false} & \text{converts into} & \text{false}, & \\
\text{false} = \text{true} & \text{converts into} & \text{true}. &
\end{array}
$$

We require that the resulting notion of (standard) reduction has the properties that

- the reduction sequence for any term r terminates with a normal form r^*,
- r^* has the same value as r, and
- any closed term of a ground type has a constant as its normal form.

These properties can be proved easily in standard examples (e.g. for the examples in section 5), by the methods of [13]. We do not try here to formulate some general criteria; cf [2] for related work.

In addition, we define the *degree of totality* $\mathrm{tdeg}(r)$ of an $\mathfrak{F}$–term r. The intention is that $\mathrm{tdeg}(r)$ should be 2 if the value $|\vec{x} \mapsto r|$ of r is "supertotal", i.e. defined on *all* arguments, 1 if $|r|$ is total, i.e. defined on *total* arguments, and 0 otherwise. So assume that for all program constants and function symbols a degree of totality is given. We then define $\mathrm{tdeg}(\hat{x}) = 0$, $\mathrm{tdeg}(x) = 1$, $\mathrm{tdeg}([u]^\rho) = 0$, $\mathrm{tdeg}(N) = \mathrm{tdeg}(R_\rho) = 1$, $\mathrm{tdeg}(\lambda \hat{x} r) = \mathrm{tdeg}(r)$,

$$
\mathrm{tdeg}(ts) = \begin{cases} 2, & \text{if } \mathrm{tdeg}(t) = 2; \\ \min(\mathrm{tdeg}(t), \mathrm{tdeg}(s)), & \text{otherwise} \end{cases}
$$

and

$$
\mathrm{tdeg}(f r_1 \ldots r_m) = \begin{cases} 2, & \text{if } f \text{ has degree } \mathrm{tdeg}_f = 2; \\ \min(\mathrm{tdeg}_f, \mathrm{tdeg}(r_1), \ldots, \mathrm{tdeg}(r_m)), & \text{otherwise,} \end{cases}
$$

where of course we assume that $\delta_{\mathrm{nat}}, \delta_{\mathrm{boole}}, =_{\mathrm{nat}}$ and $=_{\mathrm{boole}}$ have the degree of totality 2. We call a term *total* if $\mathrm{tdeg}(r) > 0$.

An atomic formula is built from a total term r of type boole, and is written $\mathrm{atom}(r)$ or just r. Falsity, i.e. $\mathrm{atom}(\mathrm{false})$, is denoted by $\bot$, and similarly truth, i.e. $\mathrm{atom}(\mathrm{true})$, is denoted by $\top$. Formulas are built from atomic formulas by $\varphi \to \psi$ and $\forall \hat{x}^\rho \varphi$ as well as $\forall x^\rho \varphi$ (whereas on the term side only $\lambda \hat{x}^\rho r$ is allowed and not $\lambda x^\rho r$). As usual we define

$$\begin{aligned} \neg\varphi &:= \varphi \to \bot, \\ \varphi \vee \psi &:= \neg\varphi \to \neg\psi \to \bot, \\ \exists x \varphi &:= \neg\forall x \neg\varphi. \end{aligned}$$

As our deductive formalism we choose Gentzen's natural deduction system, more precisely just introduction and elimination rules for $\to$ and $\forall$. For the elimination rule $\forall^-$ we have to take into account that we have two sorts of variables here, and hence we have the two rules

$$\frac{\forall \hat{x} \varphi}{\varphi[r]} \qquad \text{and} \qquad \frac{\forall x \varphi}{\varphi[r]} \quad \text{if } \mathrm{tdeg}(r) > 0.$$

In order to express the intended range of the total variables properly we also need *definedness axioms*

$$\forall x \varphi(x) \to \forall \hat{x}.\delta(\hat{x}) \to \varphi(\hat{x}).$$

The other direction can be derived from an obvious *truth axiom* $\top$, since $\delta(r)$ converts into true if $\mathrm{tdeg}(r) > 0$.

Our *induction axioms* come in two forms for each of the ground types nat and boole, since we have variables for partial and for total objects:

$$\begin{gathered} \varphi(0) \to (\forall n.\varphi(n) \to \varphi(n+1)) \to \forall n \varphi(n), \\ \varphi(\bot^{\mathrm{nat}}) \to \varphi(0) \to (\forall \hat{n}.\varphi(\hat{n}) \to \varphi(\hat{n}+1)) \to \forall \hat{n} \varphi(\hat{n}), \\ \varphi(\mathrm{true}) \to \varphi(\mathrm{false}) \to \forall p \varphi(p), \\ \varphi(\bot^{\mathrm{boole}}) \to \varphi(\mathrm{true}) \to \varphi(\mathrm{false}) \to \forall \hat{p} \varphi(\hat{p}). \end{gathered}$$

Finally we need *extensionality axioms* (i.e. axioms postulating the compatibility of extensional equality with application). To formulate them we first have to define extensional equality $=_\rho$ for an arbitrary type ρ, by

$$z_1 =_{\rho\to\sigma} z_2 :\equiv \forall \hat{x}^\rho (z_1 \hat{x} =_\sigma z_2 \hat{x}).$$

Then we require as axioms, for an arbitrary type $\rho \to \sigma$,

$$\hat{x} =_\rho \hat{y} \to \hat{z}\hat{x} =_\sigma \hat{z}\hat{y} \tag{$\mathrm{Ext}_{\rho\to\sigma}$}$$

and also for any function symbol f of functionality $(\rho_1, \dots, \rho_m) \to \sigma$

$$\hat{x}_1 =_{\rho_1} \hat{y}_1 \to \dots \to \hat{x}_m =_{\rho_m} \hat{y}_m \to f\vec{x} =_\sigma f\vec{y}. \tag{Ext_f}$$

As already mentioned, we can now use boolean induction to prove

$$\forall p. \neg\neg p \to p$$

and hence the stability $\neg\neg A \to A$ for any atomic formula A. Since

$$\begin{gathered} (\neg\neg\psi \to \psi) \to \neg\neg(\varphi \to \psi) \to \varphi \to \psi \\ (\neg\neg\varphi \to \varphi) \to \neg\neg\forall \hat{x} \varphi \to \forall \hat{x} \varphi \\ (\neg\neg\varphi \to \varphi) \to \neg\neg\forall x \varphi \to \forall x \varphi \end{gathered}$$

can be derived easily, we get the stability $\neg\neg\varphi \to \varphi$ for an arbitrary formula φ, and hence classical logic.

4. Realizing terms

We now want to argue that the formal system of higher order arithmetic introduced in the previous section is indeed capable of formalizing constructive proofs, although it does not contain a strong existential quantifier $\exists^*$. The reason is simply that a constructive proof of $\forall x \exists^* y \varphi(x, y)$ contains an algorithm to construct such a y from a given x. Now if we have a "balanced" system where the induction axioms are of the same proof theoretic strength as the principles of recursive definitions allowed, then this algorithm can in fact be formulated as a term of our language, and what we actually have proved is the "judgement" (Urteil)

$$r \quad \text{realizes} \quad \forall x \exists^* y \varphi(x, y),$$

which just means $\forall x \varphi(x, rx)$. This simple idea, which in fact is due to Weyl [15], can easily be extended to arbitrary formulas built up with $\rightarrow, \forall, \exists^*$; this is carried out below.

To summarize, what we achieve is a more explicit formulation of constructive mathematics: not just formulas containing the strong existential quantifier $\exists^*$, but judgements including realizing terms. These judgements can then be translated (interpreted) in the $\rightarrow\forall$–fragment of our higher order arithmetical language, without changing its intended meaning.

A classical proof of $\forall x \exists y \varphi(x, y)$ generally does *not* yield a program to compute y from x. The reason for this is that there might be a universal quantifier $\forall z$ right after $\exists y$, i.e. after $\neg\forall y \neg$, and this makes it possible that an assumption

$$\forall y \neg \forall z \psi(x, y, z)$$

is instantiated with a non–constant term containing critical variables which are bound later by $\forall z$.

It is well known that this is not just a technical difficulty: if T denotes Kleene's T–predicate, then

$$\forall n \exists m \forall k . T(n, n, k) \rightarrow T(n, n, m)$$

is trivially provable even in minimal logic (with $\exists m$ defined as $\neg\forall m \neg$, i.e. in classical logic), but there is no computable function f satisfying

$$\forall n, k . T(n, n, k) \rightarrow T(n, n, f(n)),$$

for then $\exists k T(n, n, k)$ would be decidable.

We now define *judgements* to be expressions of the form

$$r_1^{\rho_1}, \ldots, r_m^{\rho_m} \in \varphi$$

(to be read $r_1^{\rho_1}, \ldots, r_m^{\rho_m}$ realize φ), where φ is a formula built from atomic formulas using $\rightarrow$, $\forall$ and $\exists^*$, and $\rho_1, \ldots, \rho_m = \text{types}(\varphi)$ are a sequence of types associated with φ, defined as follows.

$$\text{types}(A) = \text{empty},$$

and if $\text{types}(\varphi) = \vec{\rho}$ and $\text{types}(\psi) = \sigma_1, \ldots, \sigma_n$ we let

$$\begin{aligned} \text{types}(\varphi \rightarrow \psi) &= \vec{\rho} \rightarrow \sigma_1, \ldots, \vec{\rho} \rightarrow \sigma_n, \\ \text{types}(\forall \hat{x}^\rho \psi(\hat{x})) = \text{types}(\forall x^\rho \psi(x)) &= \rho \rightarrow \sigma_1, \ldots, \rho \rightarrow \sigma_n, \\ \text{types}(\exists^* \hat{x}^\rho \psi(\hat{x})) = \text{types}(\exists^* x^\rho \psi(x)) &= \rho, \sigma_1, \ldots, \sigma_n. \end{aligned}$$

To give some examples, let x, y, z be of type nat. Then

$$\begin{aligned}
\text{types}(\forall x \exists^* y A(x,y)) &= \text{nat} \to \text{nat}, \\
\text{types}(\forall x \exists^* y \exists^* z A(x,y,z)) &= (\text{nat} \to \text{nat}), (\text{nat} \to \text{nat}), \\
\text{types}(\forall x \exists^* y A(x,y) \to \exists^* z A_1(z)) &= (\text{nat} \to \text{nat}) \to \text{nat}.
\end{aligned}$$

Note that $\text{types}(\varphi) = \text{empty}$ if φ is a Harrop formula (i.e. contains $\exists^*$ in premisses of $\to$ only).

For any judgement we now define its *modified realizability interpretation*, i.e. its translation in our $\exists^*$–free language of higher order arithmetic.

$$\begin{aligned}
(A(\vec{r}))^I &:\equiv A(\vec{r}), \\
(r_1, \ldots, r_n \in \varphi \to \psi)^I &:\equiv \forall \vec{\hat{x}} (\vec{\hat{x}} \in \varphi)^I \to (r_1 \vec{\hat{x}}, \ldots, r_n \vec{\hat{x}} \in \psi)^I, \\
(r_1, \ldots, r_n \in \forall \hat{x}^\rho . \psi(\hat{x}))^I &:\equiv \forall \hat{x}^\rho (r_1 \hat{x}, \ldots, r_n \hat{x} \in \psi(\hat{x}))^I, \\
(r_1, \ldots, r_n \in \forall x^\rho . \psi(x))^I &:\equiv \forall x^\rho (r_1 x, \ldots, r_n x \in \psi(x))^I, \\
(r, s_1, \ldots, s_n \in \exists^* \hat{x}^\rho \psi(\hat{x}))^I &:\equiv (s_1, \ldots, s_n \in \psi(r))^I, \\
(r, s_1, \ldots, s_n \in \exists^* x^\rho \psi(x))^I &:\equiv (s_1, \ldots, s_n \in \psi(r))^I.
\end{aligned}$$

Note that $(\vec{r} \in \varphi)^I$ does not contain $\exists^*$ any more.

5. Examples

We now want to make the abstract material treated up to now somewhat more concrete and give some examples of how the $\to \forall$–fragment of natural deduction extended by some induction schemata can actually be used to carry out some interactive proofs. Here we use an implemetation of this formal system written in SCHEME. We have chosen this language since it is rather easy then to use the built–in evaluation mechanism of SCHEME to carry out the normalisation of proofs (a technical point here is that an inverse to the evaluation is needed to make this work; cf. [1]). This in turn makes it possible to use proofs as programs.

5.1 Quotient and remainder: proofs as programs

This is an example of the proofs–as–programs paradigm. A proof of

$$\forall m, n \exists k, l. n = (m+1) * k + l \wedge l < m+1$$

is to be used as a program to divide n by $m+1$ with remainder.

In order to carry out an interactive proof we first have to set the goal. Note that $\exists$ has to be replaced by $\neg\forall\neg$ here. (`pf` means parse–formula)

```
(make-goal '?
(pf "all m,n.(all k,l.n=(m+1)*k+l -> l<m+1 -> F) -> F"))
```

The machine responds by printing

```
;?: all m,n.(all k,l.n=(m+1)*k+l -> l<m+1 -> F) -> F
```

We clearly want to use induction on n. Hence we reduce our goal to the subformula starting with $\forall n$ by assuming that we have such an m.

```
(assume 'm)

;ok, under these assumptions we have the new goal
;?-KERNEL: all n.(all k,l.n=(m+1)*k+l -> l<m+1 -> F) -> F from  m
```

We now choose to apply the appropriate induction axiom by typing `(ind)` and get as response two new goals corresponding to the base and the step case.

```
;?-KERNEL-BASE: (all k,l.0=(m+1)*k+l -> l<m+1 -> F) -> F from  m
;?-KERNEL-STEP: all n.((all k,l.n=(m+1)*k+l -> l<m+1 -> F) -> F) ->
                (all k,l.n+1=(m+1)*k+l -> l<m+1 -> F) -> F from  m
```

The last goal `?-KERNEL-STEP` is on the top of our goal stack, hence we have to work on it first. We introduce names for the bound variable n, the induction hypothesis and the premise of the remaining implication.

```
(assume 'n 'IH 'H1)

;ok, under these assumptions we have the new goal
;?-KERNEL-STEP-KERNEL: F from
;  m  n  IH:(all k,l.n=(m+1)*k+l -> l<m+1 -> F) -> F
;  H1:all k,l.n+1=(m+1)*k+l -> l<m+1 -> F
```

Here it is advisable to use the induction hypothesis to prove this goal; hence we have to introduce another goal symbol for its premise.

```
(use-with 'IH '?1)
```

```
;ok, ?-KERNEL-STEP-KERNEL can be obtained from
;?1: all k,l.n=(m+1)*k+l -> l<m+1 -> F from
;   m   n   IH:(all k,l.n=(m+1)*k+l -> l<m+1 -> F) -> F
;   H1:all k,l.n+1=(m+1)*k+l -> l<m+1 -> F
```

We can now drop the induction hypothesis, since it is not used any more.

```
(drop 'IH)

;ok, we now have the new goal
;?1-DROPPED: all k,l.n=(m+1)*k+l -> l<m+1 -> F from
;   m   n   H1:all k,l.n+1=(m+1)*k+l -> l<m+1 -> F
```

This means that we have to prove that there is no k, l for n assuming that there is no k', l' for $n + 1$. So let us assume first that we have some k, l for n:

```
(assume 'k 'l 'H2 'H3)

;ok, under these assumptions we have the new goal
;?1-DROPPED-KERNEL: F from
;   m   n   H1:all k,l.n+1=(m+1)*k+l -> l<m+1 -> F
;   k   l   H2:n=(m+1)*k+l
;   H3:l<m+1
```

At this point we have to remember how the mathematical proof would go. We have to distinguish cases here according to whether our given $l < m + 1$ is in fact $< m$ or else $= m$. In the first case we let $k' = k$ and $l' = l + 1$, and in the second case we let $k' = k + 1$ and $l' = 0$. To carry out this case distinction within our system we use

$$\forall p, m, l. l < m + 1 \to (l < m \to p) \to (l = m \to p) \to p$$

as an auxiliary lemma, which can be proved separately by induction on m and an auxiliary induction on l. (This formula corresponds to $\forall p, m, l. l < m+1 \to (l < m \vee l = m)$ with the well–known second order definition $\varphi \vee \psi :\equiv \forall p.(\varphi \to p) \to (\psi \to p) \to p$.)

A general remark is appropriate here. Kreisel and Goad [5] have argued that purely universal lemmata like this have no computational content and hence their proofs can be omitted from a proof to be used as a program. This is true, but there is an important efficiency aspect here: if we insert this lemma just as an assumption constant into our proof, then after normalization of the instantiated proof this constant will appear in huge numbers, since it cannot be reduced away. Hence we insert a complete proof `less-suc-proof` here. The second and third premise give rise to two new goals.

```
(use-with less-suc-proof 'false 'm 'l 'H3 '?< '?=)

;ok, ?1-DROPPED-KERNEL can be obtained from
;?=: l=m -> F from
;   m   n   H1:all k,l.n+1=(m+1)*k+l -> l<m+1 -> F
```

```
;   k   l  H2:n=(m+1)*k+l
;  H3:l<m+1
;?<: l<m -> F from
;   m   n  H1:all k,l.n+1=(m+1)*k+l -> l<m+1 -> F
;   k   l  H2:n=(m+1)*k+l
;  H3:l<m+1
```

Now our active goal is `?<`. We first drop the unnecessary hypothesis `H3` by `(drop 'H3)` and then introduce `l<m` as a new hypothesis `H3` by `(assume 'H3)`. So we get

```
;ok, under these assumptions we have the new goal
;?<-DROPPED-KERNEL: F from
;   m   n  H1:all k,l.n+1=(m+1)*k+l -> l<m+1 -> F
;   k   l  H2:n=(m+1)*k+l
;  H3:l<m
```

Since we are in the case `l<m` here, we proceed according to the outline above and use our hypothesis `H1` with $k' = k$ and $l' = l + 1$. Its two premises then become `n+1=(m+1)*k+l+1` and `l+1<m+1`, which are recognized by the system to be equivalent to `H2` and `H3`, respectively. Hence we type (`pt` means parse term)

```
(use-with 'H1 'k (pt "l+1") 'H2 'H3)

;ok, ?<-DROPPED-KERNEL is proved. The active goal now is
;?=: l=m -> F from
;   m   n  H1:all k,l.n+1=(m+1)*k+l -> l<m+1 -> F
;   k   l  H2:n=(m+1)*k+l
;  H3:l<m+1
```

Again we first drop the unnecessary hypothesis `H3` by `(drop 'H3)` and then introduce `l=m` as a new hypothesis `H3` by `(assume 'H3)`. So we get

```
;ok, under these assumptions we have the new goal
;?=-DROPPED-KERNEL: F from
;   m   n  H1:all k,l.n+1=(m+1)*k+l -> l<m+1 -> F
;   k   l  H2:n=(m+1)*k+l
;  H3:l=m
```

According to the outline above we now use our hypothesis `H1` with $k' = k + 1$ and $l' = 0$. Its two premises then become `n+1=(m+1)*(k+1)+0` and `0<m+1`. The second one is recognized by the system to be equivalent to `truth`, and the first one is treated as a new goal.

```
(use-with 'H1 (pt "k+1") 0 '?1 truth-axiom)

;ok, ?=-DROPPED-KERNEL can be obtained from
;?1: n+1=(m+1)*(k+1)+0 from
```

```
;  m  n  H1:all k,l.n+1=(m+1)*k+l -> l<m+1 -> F
;  k  l  H2:n=(m+1)*k+l
;  H3:l=m
;No value
```

After dropping `H1` we normalize this goal, which in this case means that the obvious arithmetic simplifications are carried out. Hence we type `(drop 'H1)` and then `(ng)` (for normalize goal) and get

```
;ok, the normalized goal is
;?1-DROPPED-NF: n=m*k+k+m from
;  m  n  k  l  H2:n=m*k+k+l
;  H3:l=m
```

We now use transitivity of equality (which again is proved separately by inductions)

```
(use-with trans-=-proof 'n (pt "m*k+k+l") (pt "m*k+k+m") 'H2 '?2)
```

```
;ok, ?1-DROPPED-NF can be obtained from
;?2: m*k+k+l=m*k+k+m from
;  m  n  k  l  H2:n=m*k+k+l
;  H3:l=m
```

This goal clearly rewrites to `l=m`, which is one of our hypotheses. This triviality at last is recognized by the system:

```
(immed)
```

```
;ok, ?2-NF is immediate from extended context. The active goal is
;?-KERNEL-BASE: (all k,l.0=(m+1)*k+l -> l<m+1 -> F) -> F from  m
```

The base case is clearly satisfied by $k = l = 0$. So we conclude our interactive proof by

```
(assume 'H1)
```

```
;ok, under these assumptions we have the new goal
;?-KERNEL-BASE-KERNEL: F from
;  m  H1:all k,l.0=(m+1)*k+l -> l<m+1 -> F
```

```
(use-with 'H1 0 0 truth-axiom truth-axiom)
```

```
;ok, ?-KERNEL-BASE-KERNEL is proved. Proof finished.
```

Since in the process of carrying out the interactive proof the system has built up internally a "partial" proof (consisting of the context, the end formula and a type free

λ–term), we have at this stage the completed proof (held in a global variable pp) available. We now normalize it (np = normalize proof)

```
(define npp (np pp))
```

This normalized proof of

$$\forall m, n \exists k, l. n = (m+1) * k + l \wedge l < m+1$$

can now be instantiated to particular numbers (e.g. 16 and 123 if we wish to divide 123 by 17) and then be normalized again.

```
(define npp123/17 (np (elim npp 16 123)))
```

The result is a rather short proof, which allows it to read off immediately the quotient and the remainder (dp = display proof)

```
(dp npp123/17)

;.....all k,l.123=(16+1)*k+l -> l<16+1 -> F by assumption u131
;....all l.123=(16+1)*7+l -> l<16+1 -> F by all-elim
;...123=(16+1)*7+4 -> 4<16+1 -> F by all-elim
;...T by truth-axiom-symbol
;..4<16+1 -> F by imp-elim
;..T by truth-axiom-symbol
;.F by imp-elim
;(all k,l.123=(16+1)*k+l -> l<16+1 -> F) -> F by ->-intro U131
```

In the second and third line k and l are instantiated by 7 (the quotient) and 4 (the remainder), respectively.

5.2 Quotient and remainder: proofs about programs

Instead of using the proof of

$$\forall m, n \exists k, l. n = (m + 1) * k + l \wedge l < m + 1$$

as a program, we also could more directly turn the algorithmic idea of the proof into a program and then prove that the values of the program at arbitrary n, m are pairs k, l with the properties required. This can be done easily in our setting, since any higher order term of our language can be evaluated on *numeric* arguments (i.e. used as a program) as well as on *variable* arguments (i.e. used in a proof dealing with variables).

A slight point to note here is that for efficiency reasons it is not advisable to define two terms here by simultaneous recursion (one for the quotient and one for the remainder), since then their evaluation leads to excessive (in fact, exponentially many) recalculations. Rather, one should define a single pair–valued function here, which we call `qr`. Accordingly we denote the left projection by `quot` and the right projection by `rem`. The recursive definition of `qr` clearly expresses the algorithmic idea of the proof above. (`c-@` = construct pair)

```
(define qr
  (lambda (n)
    (lambda (m)
      (cond ((zero-nat? n) (c-@ 0 0))
            ((suc-nat? n)
             (let* ((prev-qr ((qr (pred-nat n)) m))
                    (prev-quot (quot prev-qr))
                    (prev-rem (rem prev-qr)))
               (c-if-nat (<-nat prev-rem m)
                (c-@ prev-quot (c-+ prev-rem 1))
                (c-@ (c-+ prev-quot 1) 0))))
            (else (list (list 'qr n) m))))))
```

The goal to be proved is

```
all m,n.n=(m+1)*(quot(qr n m))+(rem(qr n m)) & (rem(qr n m))<m+1
```

We do not want to comment in detail on the interactive proof of this goal here. It can be done easily, again using induction on n and arguing by cases. In the induction step we make use of the fact that $\mathrm{qr}(n + 1, m)$ rewrites to an if–then–else term according to its definition above. The same term qr of course be used as a program and evaluated at numerical arguments, which is somewhat faster than by normalizing the instantiated proof as above.

5.3 Register: terms as hardware

Here we come back to the example from section 1, the register. Its specification was

$$\forall \hat{c}, \hat{\imath}, n, l.l < n \to \hat{c}(l) = \text{true} \to (\forall m.l < m < n \to \hat{c}(m) = \text{false}) \to \text{reg}(\hat{c}, \hat{\imath}, n) = \hat{\imath}(l). \tag{$*$}$$

The first thing to note is that if we want to apply the proofs–as–programs paradigm here, i.e. obtain a program to compute reg from a proof of

$$\forall \hat{c}, \hat{\imath}, n \exists \hat{w} \forall l.l < n \to \hat{c}(l) = \text{true} \to (\forall m.l < m < n \to \hat{c}(m) = \text{false}) \to \hat{w} = \hat{\imath}(l),$$

then a proof of this will generally *not* yield a program to compute $\hat{w}$ from $\hat{c}$, $\hat{\imath}$ and n. As dicussed in section 4, the reason for this is the universal quantifier $\forall l$ after $\exists \hat{w}$, i.e. after $\neg \forall \hat{w} \neg$, for this makes it possible that an assumption

$$\forall \hat{w} \neg \forall l \varphi(\hat{c}, \hat{\imath}, n, \hat{w}, l)$$

is instantiated with a non–constant term containing critical variables which are bound later by $\forall l$.

In our case here the situation is better since the universal quantifier $\forall l$ is in fact *bounded*, and hence the formula starting with $\forall l.l < n \to \cdots$ could be replaced by a recursively defined predicate $P(\hat{c}, \hat{\imath}, n, \hat{w})$. With this modification a proof of our specification could be used as a program. However, since we have already dealt with such an example in section 5.1, we do not pursue this matter any further here.

What we rather want to do in the present section is to treat our example according to the proofs–about–programs paradigm, for this gives an opportunity to demonstrate

- how to deal within our system with primitive recursively defined second order program constants and variables for partial functions, and
- how to translate such a constant into a circuit.

We first have to extend our language by a new program constant reg of the appropriate type. Then reg is defined (i.e. made into an operating constant) by

```
(define reg
  (lambda (c) (lambda (i) (lambda (n)
    (cond ((and (suc-nat? n) (synt-total? n))
           (let ((n-1 (pred-nat n)))
             (c-if-word (c n-1) (i n-1) (((reg c) i) n-1))))
          (else
           (list (list (list 'reg
                             (obj-to-term c (c-arrow 'nat 'boole)))
                       (obj-to-term i (c-arrow 'nat 'word)))
                 n)))))))
```

We now set the goal by

```
(make-goal '?
(pf "all n,l,c^,i^.l<n -> (c^ l)=true ->
```

```
                    (all m.l<m -> m<n -> (c^ m)=false) ->
                    (reg c^ i^ n)=(i^ l)"))
```

then type (ind) to express that we want to prove it by induction on the outermost quantified variable n. First we have to prove the induction step, which is of the form

$$\forall n.\forall l, \hat{c}, \hat{\imath}\varphi(n, l, \hat{c}, \hat{\imath}) \to \forall l, \hat{c}, \hat{\imath}\varphi(n+1, l, \hat{c}, \hat{\imath}).$$

Hence we give names to the generalized variables and premises of the induction step by

```
(assume 'n 'IH 'l 'c^ 'i^ 'H1 'H2 'H3)
```

The result is

```
;ok, under these assumptions we have the new goal
;?-STEP-KERNEL: (reg c^ i^(n+1))=(i^ l) from
;  n  IH:all l,c^,i^.l<n -> (c^ l)=true ->
         (all m.l<m -> m<n -> (c^ m)=false) -> (reg c^ i^ n)=(i^ l)
;  l  c^  i^  H1:l<n+1
;  H2:(c^ l)=true
;  H3:all m.l<m -> m<n+1 -> (c^ m)=false
```

In order to work with this goal, we first unfold the definition of reg by normalizing it, i.e. we type (ng) and get

```
;ok, the normalized goal is
;?-STEP-KERNEL-NF:
[if c^ n then i^ n else reg c^ i^ n]=(i^ l) from ...
```

Now it is rather obvious that we should distinguish cases according to the possible values of $\hat{c}(n)$. Note that $\hat{c}$ is a partial function, so undefined–boole is a possible value here, and we must have the non–strict equality between booolean terms available to deal properly with this case.

```
(cases-term (pt "c^ n"))

;?-STEP-KERNEL-NF-CASE-UNDEFINED: (c^ n)=undefined-boole ->
[if undefined-boole then i^ n else reg c^ i^ n]=(i^ l) from ...

;?-STEP-KERNEL-NF-CASE-TRUE: (c^ n)=true ->
[if true then i^ n else reg c^ i^ n]=(i^ l) from ...

;?-STEP-KERNEL-NF-CASE-FALSE: (c^ n)=false ->
[if false then i^ n else reg c^ i^ n]=(i^ l) from ...
```

The last case false is on top of our goal stack. We treat it by first first giving a name to its hypothesis by (assume 'H4) and the normalizing it by (ng):

```
;ok, the normalized goal is
;?-STEP-KERNEL-NF-CASE-FALSE-KERNEL-NF: (reg c^ i^ n)=(i^ l) from
;  n  IH:all l,c^,i^.l<n -> (c^ l)=true ->
         (all m.l<m -> m<n -> (c^ m)=false) -> (reg c^ i^ n)=(i^ l)
;  l  c^  i^  H1:l<n+1
;  H2:(c^ l)=true
;  H3:all m.l<m -> m<n+1 -> (c^ m)=false
;  H4:(c^ n)=false
```

Here we clearly can use the induction hypothesis. Its first and third premise are introduced as new goals, and its second premise is H2.

```
(use-with 'IH 'l 'c^ 'i^ '?1 'H2 '?2)

;ok, ?-STEP-KERNEL-NF-CASE-FALSE-KERNEL-NF can be obtained from
;?2: all m.l<m -> m<n -> (c^ m)=false from ...
;?1: l<n from
;  n  IH:...
;  l  c^  i^  H1:l<n+1
;  H2:(c^ l)=true
;  H3:all m.l<m -> m<n+1 -> (c^ m)=false
;  H4:(c^ n)=false
```

To conclude $l < n$ from $l < n + 1$ it suffices to exclude the case $l = n$. We do that by adding a lemma expressing that fact as a global assumption (it could as well be proved easily by induction on n) by (aga = add global assumption)

```
(aga 'cases-suc (pf "all l,n.(l=n -> F) -> l<n+1 -> l<n"))
```

and then use it with its first premise taken as a new goal:

```
(use-with 'cases-suc 'l 'n '?3 'H1)

;ok, ?1 can be obtained from
;?3: l=n -> F from ... H2:(c^ l)=true H4:(c^ n)=false
```

It is rather obvious how we should proceed here. We first give a name to the premise $l = n$ by (assume 'H5). Since true=false rewrites to absurdity F, we can reduce our goal via transitivity and symmetry of boolean equality (between possibly undefined objects!) to $\hat{c}(l) = \hat{c}(n)$. Hence we type

```
(aga 'lemma1
(pf "all p^1,p^2,p^3,p^4.p^1=p^2 -> p^1=p^3 -> p^2=p^4 -> p^3=p^4"))

(use-with 'lemma1 (pt "c^ l") (pt "c^ n") true false '?4 'H2 'H4)

;ok, ?3-KERNEL can be obtained from
```

```
;?4: (c^ l)=(c^ n) from ... H5:l=n
```

Here we clearly need an equality axiom for $\hat{c}$:

```
(aga 'comp-c^ (pf "all c^,n1,n2.n1=n2 -> (c^ n1)=(c^ n2)"))

(use-with 'comp-c^ 'c^ 'l 'n 'H5)

;ok, ?4 is proved. The active goal now is
;?2: all m.l<m -> m<n -> (c^ m)=false from ...
;  H3:all m.l<m -> m<n+1 -> (c^ m)=false
```

After introducing names for the variables/assumptions with `(assume 'm 'H5 'H6)` we clearly have to use `H3` with its second premise introduced as a new goal.

```
(use-with 'H3 'm 'H5 '?5)

;ok, ?2-KERNEL can be obtained from
;?5: m<n+1 from ...  H6:m<n
```

This can be proved from transitivity of $<$, making use of the fact that $n < n+1$ rewrites to `true`.

```
(aga 'trans-less (pf "all n1,n2,n3.n1<n2 -> n2<n3 -> n1<n3"))

(use-with 'trans-less 'm 'n (pt "n+1") 'H6 truth-axiom)

;ok, ?5 is proved. The active goal now is
;?-STEP-KERNEL-NF-CASE-TRUE: (c^ n)=true ->
[if true then i^ n else reg c^ i^ n]=(i^ l) from ...
```

Again we introduce a name for the hypothesis by `(assume 'H4)` and then normalize the goal by `(ng)`.

```
;ok, the normalized goal is
;?-STEP-KERNEL-NF-CASE-TRUE-KERNEL-NF: (i^ n)=(i^ l) from
;  n  IH:all l,c^,i^.l<n -> (c^ l)=true ->
         (all m.l<m -> m<n -> (c^ m)=false) -> (reg c^ i^ n)=(i^ l)
;  l  c^  i^  H1:l<n+1
;  H2:(c^ l)=true
;  H3:all m.l<m -> m<n+1 -> (c^ m)=false
;  H4:(c^ n)=true
```

Here we proceed as follows. From $l < n + 1$ we can conclude that either $l < n$ or else $l = n$. The first case is impossible since then from `H3` we get $\hat{c}(n) = \text{false}$ contradicting `H4`, and in the second case the goal simply follows from an equality axiom. For brevity we just state what the user has to type in order to carry out this proof plan (i.e. his "tactic") and leave out most of the the system responses.

```
(aga 'less-suc
     (pf "all n,l,p.l<n+1 -> (l<n -> p) -> (l=n -> p) -> p"))

(use-with
 'less-suc
 'n 'l (app '=-word (pt "i^ n") (pt "i^ l")) 'H1 '?< '?=)

;ok, ?-STEP-KERNEL-NF-CASE-TRUE-KERNEL-NF can be obtained from
;?=: l=n -> (i^ n)=(i^ l) from ...
;?<: l<n -> (i^ n)=(i^ l) from ...

(assume 'H5)
```

According to our proof plan we want to argue that this case cannot happen. Hence we use ex–falso–quodlibet here, i.e. the lemma $\forall p.\bot \to p$ (which could be proved easily by boolean induction on p)

```
(use-with 'efq (app '=-word (pt "i^ n") (pt "i^ l")) '?6)

;ok, ?<-KERNEL can be obtained from
;?6: F from ...
;  H3:all m.l<m -> m<n+1 -> (c^ m)=false
;  H4:(c^ n)=true
;  H5:l<n

(aga 'lemma2 (pf "all p^1,p^2,p^3.p^1=p^2 -> p^1=p^3 -> p^2=p^3"))

(use-with 'lemma2 (pt "c^ n") true false 'H4 '?7)

;ok, ?6 can be obtained from
;?7: (c^ n)=false from ...
;  H3:all m.l<m -> m<n+1 -> (c^ m)=false
;  H5:l<n

(use-with 'H3 'n 'H5 truth-axiom)

;ok, ?7 is proved. The active goal now is
;?=: l=n -> (i^ n)=(i^ l) from ...

(assume 'H5)

(aga 'comp-i^ (pf "all l,n.l=n -> (i^ n)=(i^ l)"))
```

```
(use-with 'comp-i^ 'l 'n 'H5)

;ok, ?=-KERNEL is proved. The active goal now is
;?-STEP-KERNEL-NF-CASE-UNDEFINED: (c^ n)=undefined-boole ->
[if undefined-boole then i^ n else reg c^ i^ n]=(i^ l) from ...

(assume 'H4)

(ng)

;ok, the normalized goal is
;?-STEP-KERNEL-NF-CASE-UNDEFINED-KERNEL-NF: uw=(i^ l) from ...
;  H1:l<n+1
;  H2:(c^ l)=true
;  H3:all m.l<m -> m<n+1 -> (c^ m)=false
;  H4:(c^ n)=undefined-boole
```

Here we proceed in a similar way as in the case true above. From $l < n + 1$ we can conclude that either $l < n$ or else $l = n$. The first case is impossible since then form `H3` we get $\hat{c}(n) =$ false contradicting `H4`, and the second case $l = n$ is impossible too since then `H2` and `H4` lead to a contradiction.

```
(use-with
 'less-suc
 'n 'l (app '=-word 'uw (pt "i^ l")) 'H1 '?u< '?u=)

;ok, ?-STEP-KERNEL-NF-CASE-UNDEFINED-KERNEL-NF can be obtained from
;?U=: l=n -> uw=(i^ l) from ...
;?U<: l<n -> uw=(i^ l) from ...

(assume 'H5)

(use-with 'efq (app '=-word 'uw (pt "i^ l")) '?8)

;ok, ?U<-KERNEL can be obtained from
;?8: F from ...
;  H3:all m.l<m -> m<n+1 -> (c^ m)=false
;  H4:(c^ n)=undefined-boole
;  H5:l<n

(use-with 'lemma2 (pt "c^ n") 'undefined-boole false 'H4 '?9)

;ok, ?8 can be obtained from
;?9: (c^ n)=false from ...

(use-with 'H3 'n 'H5 truth-axiom)
```

```
;ok, ?9 is proved. The active goal now is
;?U=: l=n -> uw=(i^ l) from ...

(assume 'H5)

;ok, under these assumptions we have the new goal
;?U=-KERNEL: uw=(i^ l) from ... H5:l=n

(use-with 'efq (app '=-word 'uw (pt "i^ l")) '?10)

;ok, ?U=-KERNEL can be obtained from
;?10: F from ...
;  H2:(c^ l)=true
;  H4:(c^ n)=undefined-boole
;  H5:l=n

(aga  'lemma3 (pf "all p^1,p^2,p^3,p^4.
                   p^1=p^3 -> p^2=p^4 -> p^1=p^2 -> p^3=p^4"))

(use-with
'lemma3 (pt "c^ l") (pt "c^ n") 'true 'undefined-boole 'H2 'H4 '?11)

;ok, ?10 can be obtained from
;?11: (c^ l)=(c^ n) from ... H5:l=n

(use-with 'comp-c^ 'c^ 'l 'n 'H5)

;ok, ?11 is proved. The active goal now is
;?-BASE: all l,c^,i^.l<0 -> (c^ l)=true ->
  (all m.l<m -> m<0 -> (c^ m)=false) -> (reg c^ i^ 0)=(i^ l)
```

Since $l < 0$ rewrites to absurdity, this goal can be proved trivially and we can just type `(immed)` to get

```
;ok, ?-BASE is immediate by ex-falso-quodlibet. Proof finished.
```

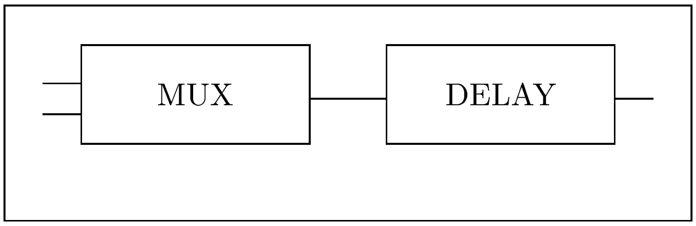
We now come to the final point we want to mention here, namely how to translate our (higher order) primitive recursive definition of reg into a circuit. We already noted that from this definition we only needed the last (recursion) equation to prove the specification given in section 1. The special form of this equation we want to make use of is that $\mathrm{reg}(\hat{c}, \hat{\imath}, n+1)$ is defined explicitely from $\mathrm{reg}(\hat{c}, \hat{\imath}, n)$ and $\hat{c}(n)$, $\hat{\imath}(n)$, in our case by plugging these three terms into an if–form, i.e.

$$\mathrm{reg}(\hat{c}, \hat{\imath}, n+1) = \mathrm{if}(\hat{c}(n), \hat{\imath}(n), \mathrm{reg}(\hat{c}, \hat{\imath}, n))$$

So the components we need are an ALU (Arithmetic–Logic–Unit) corresponding to if, usually denoted by

(for multiplexer), and a delay unit

whose output at time $n+1$ is its input at time n. Hence our circuit for the register is



Bibliography

1. Berger, U. and H. Schwichtenberg: An inverse of the evaluation functional for typed λ–calculus. In E. Vemuri (ed.): Proc 6th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, Los Alamitos 1991, pp. 203–211
2. Breazu–Tannen, V. and J.Gallier: Polymorphic rewriting conserves algebraic strong normalization. Theoretical Computer Science *83*, 3–28 (1991)
3. Ershov, Yu.L.: Model C of partial continuous functionals. In R. Gandy and M. Hyland (eds.): Logic Colloquium 1976. North Holland, Amsterdam 1977, pp. 455–467
4. Friedman, H.: Equality between functionals. In R. Parikh (ed.): Logic Colloquium, Lecture Notes in Math 453. Springer, Berlin 1975, pp. 22–37
5. Goad, C.: Computational uses of the manipulation of formal proofs. Stanford Department of Computer Science, Report No. STAN–CS–80–819, 1980
6. Gödel, K.: Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. Dialectica *12*, 280–287 (1958)
7. Kreisel, G.: Interpretation of analysis by means of constructive functionals of finite types. In A. Heyting (ed.): Constructivity in Mathematics. North Holland, Amsterdam 1959, pp. 101–128
8. Longo, G. and E. Moggi: The hereditatily partial recursive functionals and recursion theory in higher types. The Journal of Symbolic Logic *49* (4), 1319–1332 (1984)
9. Plotkin, G.D.: LCF considered as a programming language. Theoretical Computer Science *5*, 223–255 (1977)
10. Plotkin, G.D.: $\mathbf{T}^\omega$ as a universal domain. Journal of Computer and System Sciences *17*, 209–236 (1978)

11. Scott, D.S.: Domains for denotational semantics. In M. Nielsen, E.M. Schmidt (eds.): Automata, Languages and Programming. Lecture Notes in Computer Science 150, Springer, Berlin 1982, pp. 577–613

12. Schwichtenberg, H.: A normal form for natural deductions in a type theory with realizing terms. In: Atti del Congresso Logica e Filosofia della Scienza, oggi. San Gimignano 1983. Vol. I – Logica. CLUEB, Bologna 1986, pp. 95–138.

13. Schwichtenberg, H.: Primitive recursion on the partial continuous functionals. In M. Broy (ed.): Informatik und Mathematik. Springer, Berlin 1991, pp. 251–268

14. Statman, R.: Equality between functionals revisited. In L.A. Harrington et al. (eds.): Harvey Friedman's Research on the Foundations of Mathematics. North Holland, Amsterdam 1985, pp. 331–338

15. Weyl. H.: Über die neue Grundlagenkrise der Mathematik. Mathematische Zeitschrift *10*, 1921